For: dwight
Printed on: Fri, Jul 16, 1993 10:04:22
From book: guide
Document: ch5
Last saved on: Fri, Oct 2, 1992 15:05:27

5 Program Communication: PCS

The PCS libraries provide calls for developing data communication applications that run on iWarp systems. This chapter presents the following:

- PCS concepts
- using the data movement library
- using the array combiner library
- guidelines for using PCS
- PCS example

For more information on developing PCS applications, see Chapter 7, “Developing iWarp Programs.”

PCS Concepts

This section describes the following PCS concepts:

- PCS libraries
- cell programs
- array programs
- cell networks
- cell naming conventions
- array modules
- combining array modules
- auto-placement and auto-routing
- PCS and messages

PCS Libraries

There are two PCS libraries:

- the data movement library
- the array combiner library

You use the PCS data movement library for developing the cell programs that run on the cells of an iWarp system. You use the array combiner library for developing array programs that configure your PCS application.

This chapter does not contain information about the specific parameters for the PCS library calls. You can obtain this information from the PCS library manual pages. These manpages are available online and in Appendix C, “PCS Libraries.”

Cell Programs

Cell programs are iWarp application programs that run in parallel on the cells of an iWarp array. Cell programs can communicate with cell programs running on other iWarp cells, move data between cells, process data, and create and destroy dynamic networks between cells at runtime. For more information about dynamic networks, see page 5-21.

You create cell programs using the PCS data movement library and the PCS tools. Include the *pcs/pcs_def.h* file in the source code for your cell program to access the data movement library calls.

You compile cell programs using the iWarp C compiler (iwcc) and combine cell programs using the PCS array combiner (iwcb) to create array modules. For information about using PCS tools to develop iWarp applications, see Chapter 6, "PCS Tools."

Cell programs use ports to communicate with other cells. Ports are data structures that provide connections to networks, contain cell connection and state information, and provide access to other cells.

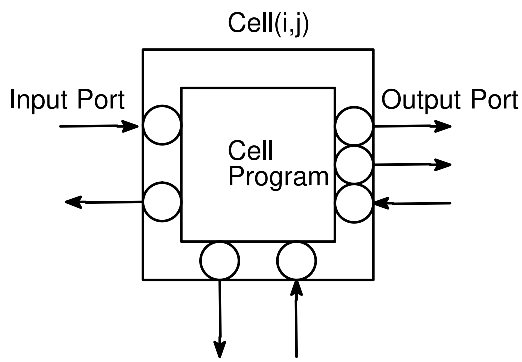

Figure 5-1: Cell Program

Figure 5-1 shows a cell program running on a cell. The cell program has both input and output ports to communicate with other cells.

Array Programs

An array program runs on the host system and configures an array module to run on an iWarp array. You use an array program to do the following:

- create ports for cell programs
- map cell programs to array cells
- configure networks

The PCS array combiner library provides the calls you use to create array programs. You include the *pcs/cb_def.h* file in the source code for your array program to access the array combiner library calls.

You submit the array program to the PCS array combiner (iwcb). The array combiner compiles and runs the array program to create the array modules for your application.

For a description of the PCS array combiner, see Chapter 6, "PCS Tools."

Cell Naming Conventions

An iWarp system has two kinds of cells: array cells and auxiliary cells. Array cells are the cells that physically reside on the iWarp array. Array cells are viewed in PCS as a grid or logical torus. Array cells are addressed in PCS using cartesian coordinates that represent the row and column in the logical array grid. You use the cartesian coordinates in cell programs and the array programs.

Auxiliary cells are additional cells in the system. They are not physically part of an iWarp array. For example, auxiliary cells on an SIB or can be disk interface cells. They are attached to the edge of the physical array, and their actual location is system dependent. Array programs and cell programs address the auxiliary cells using –1 for either a row or column designation. Naming system cells is system dependent, but a typical system with one SIB cell attaching an array to a host names the SIB as cell (0,–1).

Figure 5-2 shows an example of a logical set of addresses for an iWarp array.

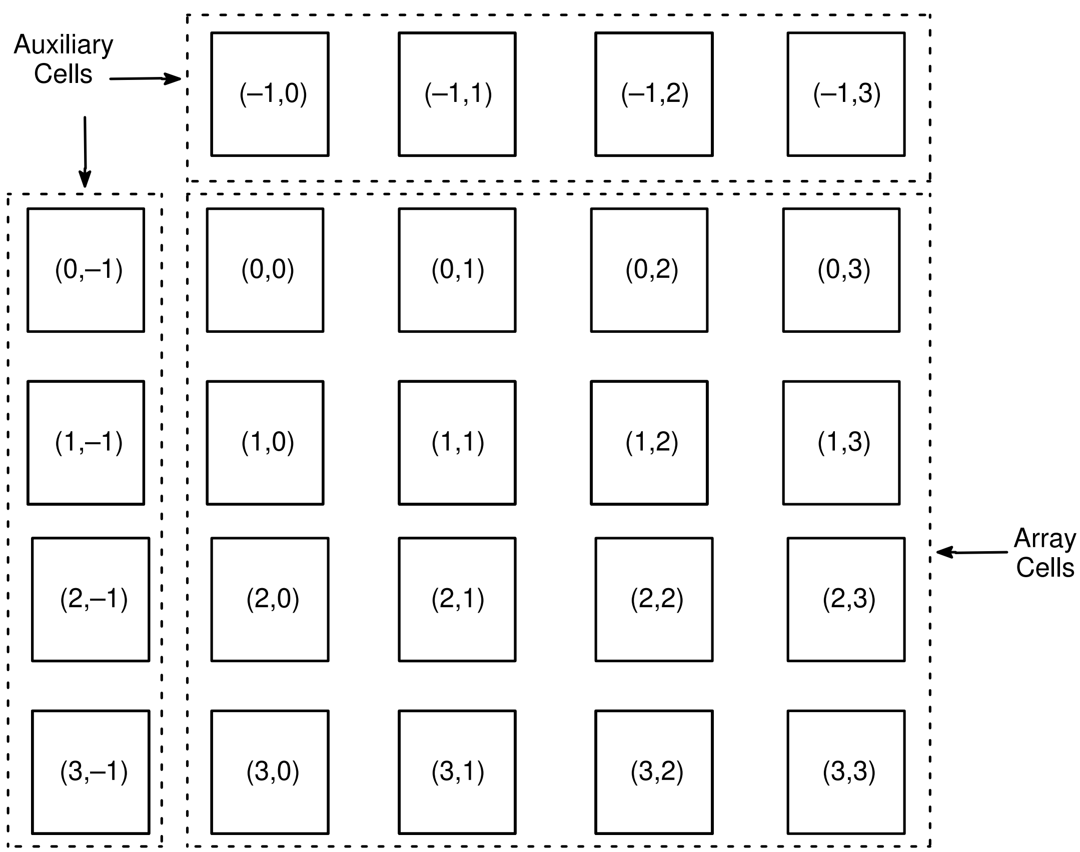

Figure 5-2: Example 4 x 4 Array with 4 Auxiliary Cells

Networks

iWarp cells can be connected together by a data communication path to create a network. PCS applications run on the cells in a network. Networks allow the cell programs to communicate, move data, and process data in parallel. You write an array program to configure cell networks. The PCS array combiner compiles and runs the array program. This creates the networks for your application. For more information about creating networks, see page 5-22.

A network is a simple path or route through two or more cells. A multi-drop network has a route through multiple cells. The endpoint cells, by default, can read from or write to the network. Intermediate cells can tap into the network to read from or write to the network. A point-to-point network is a network between two cells, without any intermediate cells tapped in.

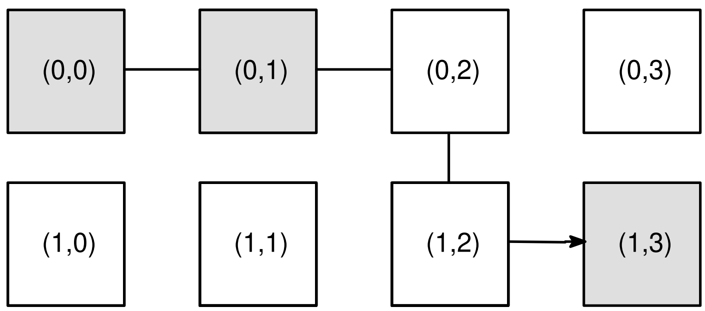

Figure 5-3: Three-Cell Application

Figure 5-3 shows three-cell application that is a multi-drop network. The network begins at cell(0,0) and ends at cell(1,3). The network has one intermediate cell that is tapped in, cell(0,1). The cells in this network are not necessarily contiguous, but can communicate directly with each other through a route.

You can also create dynamic networks at runtime. You create and destroy dynamic networks in cell programs. They have the same capabilities as static networks that you create in array programs. Dynamic networks exist only during the runtime of the cell programs that create them. For more information about creating dynamic networks, see page 5-21.

Array Modules

An array module is an execution unit that can execute on an iWarp system. An array module consists of one or more cell programs and existing array modules that you combine to create your executable application. You combine array modules using the PCS array combiner. You use the PCS array loader to load and run an array module on an iWarp system.

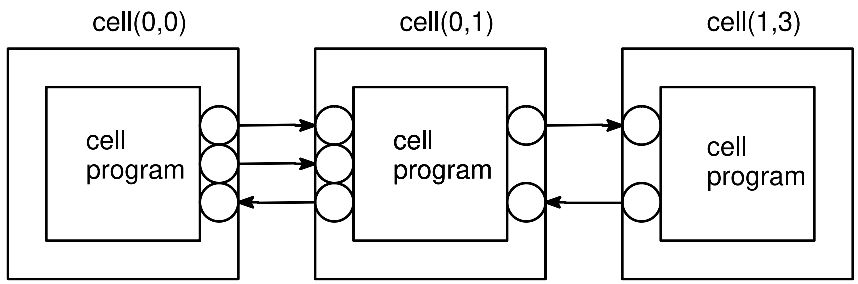

Figure 5-4: Array Module

Figure 5-4 shows an array module that is loaded onto three cells on an iWarp array. This array runs on five networks, which the direction arrows indicate.

Combining Array Modules

You can can combine multiple array modules to create new array modules. There are two methods for combining array modules: overlaying and nesting. The difference between these methods is how cells are addressed in cell programs within a particular module.

Overlay Method

In the overlay method, an array module is included in another array module. The cell programs in the included array module use the cell naming of the array module in which it is included. Figure 5-5 shows an array module that is overlayed on another array module.

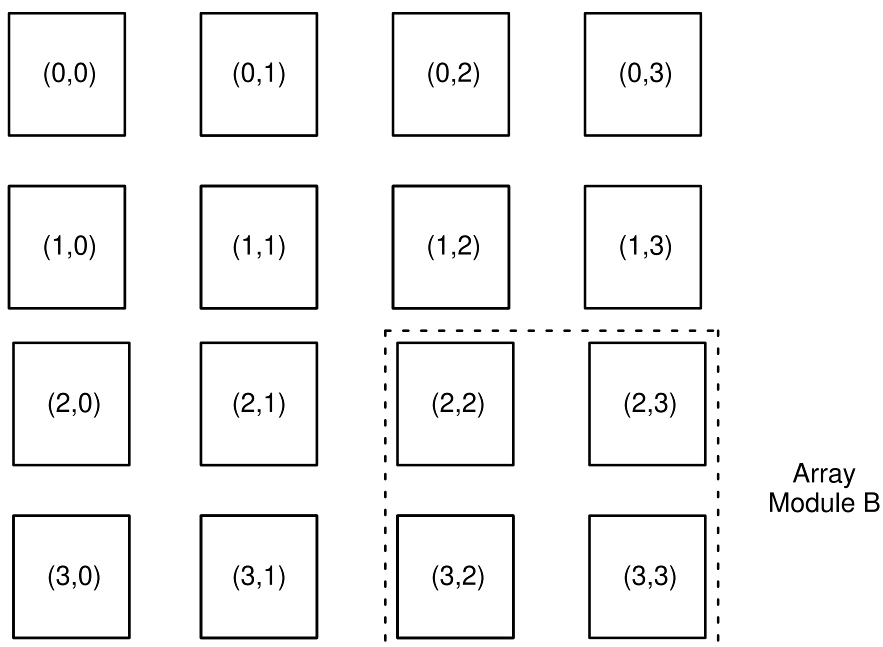

Figure 5-5: Array Module B Overlayed on a 4 x 4 Array

Figure 5-5 shows array module overlayed onto a 4 x 4 array. Array module B is placed at cell (2,2) in the array. In the overlay method, the cell program that runs on cell (0,0) in array module B now runs on cell (2,2). This method is useful for loading generic programs like a synchronization program or a program for loading predefined networks.

Nesting Method

The nesting method uses array modules as building blocks. When an array module is included into another array module, the cells in the included array module maintain their original logical cell IDs.

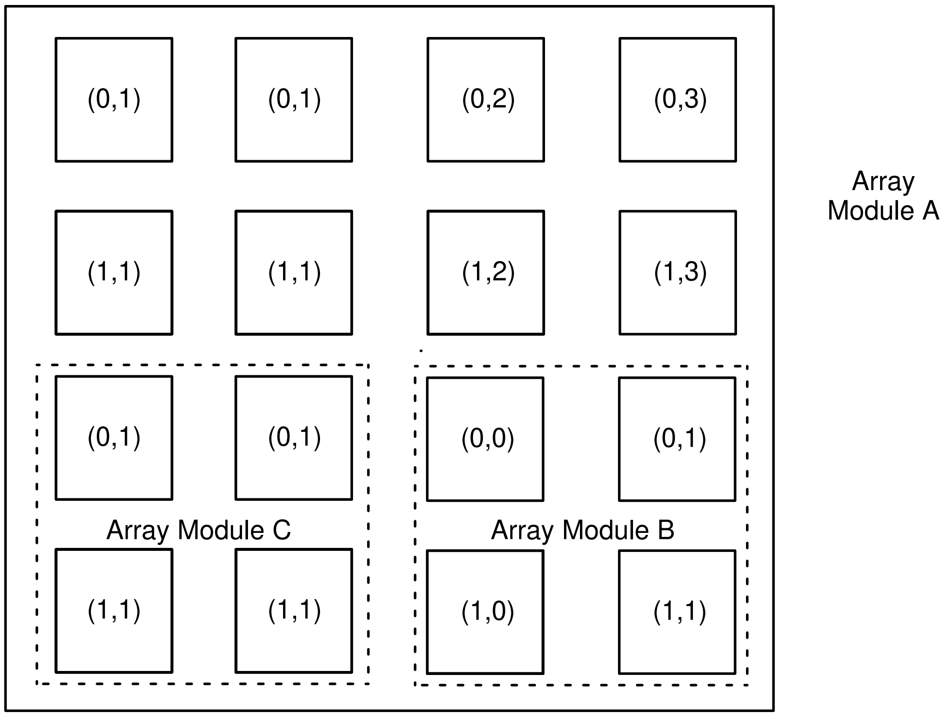

Figure 5-6: Three Nested Array Modules

Figure 5-6 shows array modules B and C nested in array module A. The nested modules retain their cell IDs. For example, in array module B, the cell(0,0) retains its cell ID. If that cell program on cell(0,0) sends data to the cell to its right, it still sends to cell (0,1). In this method the cell program on a cell uses the logical cell ID of the most deeply nested array module that encloses the cell.

External ports provide access between modules in a nested array module. Cell programs use external ports when communicating with a program outside its local array module. By connecting external ports, you can pipeline different applications together. For example, you could pipeline a matrix operation application, a signal processing application, and an image processing application together.

Auto–Placement and Auto–Routing

Instead of configuring a cell network in the array program, you can let PCS configure your cell network for you. You do this by specifying a general cell network configuration in the array program. When you combine your application, the PCS array combiner automatically places the cell programs on the array and routes the cell network for your application. The array combiner attempts to optimize the configuration.

You can use auto-placement and auto-routing with point to point networks, but not on a logical torus (using the back edges of the array). For more information about auto-placement and auto-routing, see Chapter 7 “Developing PCS Programs.”

PCS and Data Movement

PCS provides the following support for data movement on iWarp systems:

- two data movement capabilities: systolic communication and spooling
- two network modes: local mode and express

Messages in PCS

When using PCS to move data between cells in a network, you use a message format that consists of the following:

- an open message marker indicating the beginning of the message
- a sequence of data items
- a close message marker indicating the end of the message.

Figure 5-7 shows the message format for moving data in networks.

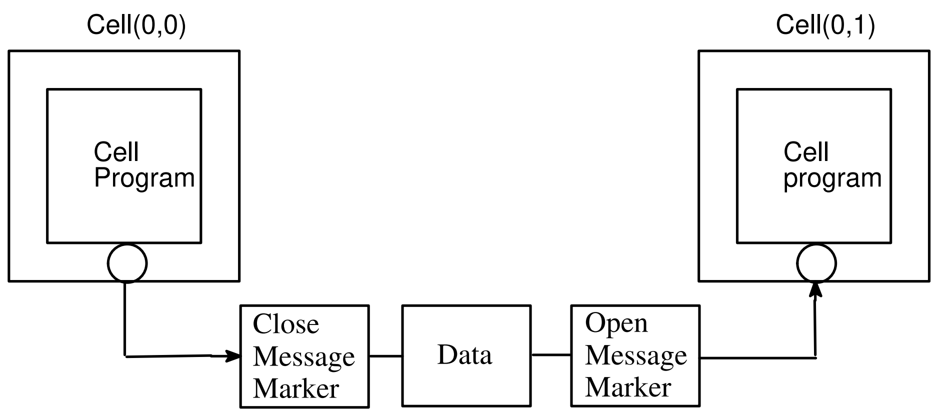

Figure 5-7: Message Format

To send and receive data between cells, you use the following sequence:

1. The sending cell sends an open message marker to receiving cell.
2. The receiving cell receives the open message marker.
3. The sending cell sends the data. The data can be data items or a data block.
4. The receiving cell receives the data.
5. The sending cell sends a close message marker.
6. The receiving cell receives the close message marker.

The PCS data movement library provides calls for using messages in your iWarp applications. For more information about messages, see the descriptions later in this chapter about sending and receiving data.

Systolic Communication

In systolic communication, the sending cell program sends a message and the receiving cell program receives the message data directly, without going through memory. The sending cell program sends an open message marker, sends the message data (item by item), and closes the message by sending a close message marker. The receiving cell program receives an open message marker, receives the message data, and receives a close message marker.

PCS supports both memory and systolic message passing. In PCS, the sending cell and the receiving cell do not need to use the same communication method. One cell program can send a message systolically and another cell can receive the message in memory.

Network Modes

A network can use one of two data movement modes: local or express. When the network is in local mode at a cell, the cell program can read from and write to the network. When in express mode, data passes through the network directly to the cell it is addressed to. A cell program does not access the data on the network, unless the data is addressed to the cell.

Cells on point-to-point networks or endpoints of other networks are always in local mode. Networks on other cells are in express mode, by default, so data can pass through them. A cell switches from express to local mode when a message header addressed to it arrives or when a stop condition is met. For more information on stop conditions, see page 5-18. A cell returns to express mode when a close message marker is sent or received or when the cell program makes a skip_msg call that explicitly sets the express mode.

Spooling Data

Spooling lets a cell program use background processes to send and receive data in your applications. The cell program can send and receive data across the network without having to wait for the data transaction to complete. During spooling, the computation agent starts the send or receive operation, but the communication agent takes over to complete the transaction. This frees the computation agent to do other data processing.

PCS provides special calls and data structures that support spooling. For more information about spooling, see page 5-16. For a spooling example, see Chapter 7, "Developing iWarp Programs."

NOTE

Use of spooling can cause programs that contain multiclock operations as the third word of a compute-and-access instruction to be unreliable. For a complete description of this problem and associated programming suggestions, please refer to the `spooling` manual page.

PCS and PathLib

PCS is the high-level communication interface for iWarp programming; PathLib is the low-level communication interface for iWarp programming. You can write PCS programs that perform the same operations as PathLib programs. But, PCS provides an interface and tools for automatically combining multiple cell programs to create cell program networks. The PCS interface is written using the PathLib library. PCS provides a preprocessor that lets you generate inline code from your PCS source, to optimize your application. For more information about the PCS preprocessor, see Chapter 6 "PCS Tools."

Using the Data Movement Library

Use the PCS data movement library in your cell programs to do the following:

- manage ports
- initialize networks
- send data to other cells
- receive data from other cells
- spool data
- preview data
- use express mode
- synchronize network operations
- test network status
- manage dynamic networks at runtime

You must include the *pcs/pcs_def.h* file in the source code of your cell program to access the data movement calls.

When data movement calls return an error status, the calls set the pcs_errno global variable with an error status.

Using Addresses in PCS Calls

Some PCS calls require an address as a parameter. Parameter addresses must be word aligned, because nonaligned addresses can cause errors with undefined results.

Managing Ports

Ports provide cell programs access to communication paths between iWarp cells. Cell programs send and receive data through ports. Cell programs can have multiple ports. There are input ports for receiving data and output ports for sending data.

Before using a port, you bind the port to a gate. A gate provides an interface between the communication agent and the computation agent and allows the port to communicate with other cells in a network.

There are two kinds of gates:

- systolic gates
- spooling gates

Systolic Gates

Systolic gates are for systolic data movement between cells in a network. There are both input and output systolic gates. A systolic input gate receives one data item at a time from the network. A systolic output sends one data item at a time across the network to another cell. A cell program can immediately use data received by a systolic input gate. There are four systolic gates on a cell:

- GATE0_IN
- GATE0_OUT
- GATE1_IN
- GATE1_OUT

You can refer to GATE0_IN and GATE0_OUT as GATE0. You can refer to GATE1_IN and GATE1_OUT as GATE1.

Spool Gates

Spool gates are for spooling data in and out of cells in the network. A cell program can use up to six spool gates. You can configure each spool gate as either an input or output gate. Spool gates can be used to move data only in one direction at a time.

Binding Gates

The following calls for binding ports to gates:

bind_systolic_gate()	Bind a port to a systolic gate.
unbind_systolic_gate()	Unbind a port from a systolic gate.
bind_spool_gate()	Bind a port to a spool gate and specify the configuration for the gate.
unbind_spool_gate()	Unbind a port from a spool gate.

Before you can use a gate or a port in a data movement library call, bind the gate to a port. A bind call unbinds a gate from any port it is currently bound to before binding it to a new port.

Initializing Networks

Before a cell program can run, it must first initialize its networks and ports for data movement operations. After a cell program completes it operations, it must close the networks and free the resources it uses. The following calls initialize and de-initialize a cell program:

pcs_init()	Initialize the networks and ports. This call reads the initialization tables the array loader creates for the network that the current cell program is connected to. The call uses the information in these tables to configure logical channels, initialize ports, and open input and output connections. Make this call before making any other data movement library calls. More debugging information will be printed if the system variable Debug_de_init is set to TRUE when pcs_init() is called.
pcs_de_init()	Closes down currently defined networks and free reserved resources. No other PCS functions should follow this call. More debugging information will be printed if the system variable Debug_de_init is set to TRUE when pcs_de_init() is called.

Getting Array Module Information

A cell program can get the following information about the array module that contains it:

- the logical cell ID of the current cell
- the size of the logical array
- the instance pointer for the array module that contains the cell program
- the instance pointer for an array modules parent
- the name of the array module that contains the cell program
- the pointer to a specific port
- the equality of two cell IDs
- the equality of the instance IDs for two array modules
- verification that a cell is an auxiliary cell
- the pointer to a cell

The following calls let a cell program get array module information:

get_cellid()	Get the cell ID for the current logical cell.
get_array_size()	Get the row and column size for the current logical array.
get_instance()	Get a pointer to the most deeply nested array module that contains the current cell program.
get_parent_instance()	Get a pointer to the array module instance that encloses the specified array module instance.
get_instance_name()	Get a pointer to the name of the array module that contains the cell program.
get_port()	Get a pointer to a requested port.

PtEqual()	Compares two cell IDs for equality.
InstanceEqual()	Compares the instance IDs of two array modules.
auxiliary()	Return TRUE is the cell is of type AUX.
cell()	Allocate space for a cell ID, initialize the cell ID, and return a pointer to it.

Sending Data

A cell program communicates with other cell programs by sending and receiving data through ports. There are two methods for sending data: systolic data movement and spooling. Systolic data movement sends data as a stream of data items. Spooling uses background processes to send data. For more information about spooling, see page 5-16.

Opening and Closing a Send Operation

To send data to another cell, you do the following:

1. send an open message to begin the send operation
2. send the data
3. send a close message to end the send operation

Use this sequence to send data on a multiple sender-receiver network (multi-drop network). This sequence is not necessary for a point-to-point network, because data movement is done directly from the sender to the receiver. In point-to-point networks, you do not have to send and receive message markers.

Use the following calls to open or close a data path for sending data between two cells on a network:

send_open_msg()	Notify the destination cell that the source cell is sending data.
send_close_msg()	Notify the destination cell that the source cell is done sending data. When an error occurs closing a message, the call returns FALSE and sets the pcs_errno variable with error information.

Sending Data Items

You can send data in the form of messages, data blocks, markers, or data words. Data words can be one of the following types: integer, float, or double. There is a separate call for each kind of data you can send. Use the following calls to send data:

send_msg()	Send a message to a destination cell.
send_block()	Send a data block to a destination cell. This appends the block to an opened message. Unless the data is sent on a point-to-point network, you must precede this call with a send_open_msg or send_direct_open_msg call. Not sending an open message can cause your program to hang.
send_marker()	Send a marker to a destination cell. Markers can be used to mark a place in a message, for example, the end of a data item or the end of a message.
send_int()	Send a 4-byte item of type integer to a destination cell.
send_float()	Send a 4-byte item of type float to a destination cell.
send_double()	Send a 8-byte item of type double to a destination cell.

You can send or receive only a number bytes that are a multiple of the system's word size.

Sending Direct Messages

For networks with multiple senders and receivers, you need to check the network for busy status before sending data. For point–to–point networks that have only one sender between cells in a network, you can send data directly from one cell to another. This method is more efficient, because the calls do not check for network status. Use the following calls to send direct messages:

send_direct_open_msg()	Send a message header to a destination cell without checking for network status.
send_direct_close_msg()	Close an opened message sent to a destination cell without checking for network status.
send_direct_msg()	Send a message to a destination cell without checking for network status.

Receiving Data

To receive data in a cell program, you request to receive the data from an input port and a gate. Before receiving data through an input port, bind the port to an input gate.

In a point-to-point network between a pair of cells, data is received in the order sent. If there are multiple senders and receivers, messages do not necessarily arrive in the order sent.

Opening and Closing Receive Operations

To receive data from another cell, you do the following:

1. receive an open message to begin the receive operation
2. receive the data
3. receive a close message to end the receive operation

Use this sequence to receive data on a multiple sender-receiver network (multi-drop network). This sequence is not necessary for a point-to-point network, because data movement is done directly from the sender to the receiver. In point-to-point networks, you do not have to send and receive message markers.

Use the following calls, to receive open and close messages from a source cell, when receiving data.

receive_open_msg()	Notifies the destination cell that data was sent and can be received. Returns TRUE for success and FALSE for failure.
receive_close_msg()	Notifies the destination cell that the data movement is done. This closes the message path. Returns TRUE for success and FALSE for failure.

Receiving Data Items

You can receive data in the form of messages, data blocks, markers, or data words. Data words can have one of the following types: integer, float, or double. There is a separate receive call for each kind of data. You use the following calls to receive data.

receive_msg()	Receive a message and place it at a specific memory location. Returns TRUE for success and FALSE for failure.
receive_block()	Receive a data block of a specified size and place it at a specific memory location. A receive_open_msg call must precede this call, unless the call is for point-to-point communication. This call checks for a close message marker. If a marker is received, the call reads the bytes available before the marker.
receive_marker()	Receive a marker. Markers can be used to mark a place in a message, such as, the end of a data item or the end of a message.
receive_int()	Receive a 4-byte item of type integer.
receive_float()	Receive a 4-byte item of type float.
receive_double()	Receive an 8-byte item of type double.

Receiving Limited Data

You can use the following two calls to receive a specific number of bytes. These calls are useful when you know the exact size of the data you are receiving. These calls are more efficient, because they do not have the overhead of checking each data item received for a close message marker.

receive_limited_msg()	Receive a message of an exact size in bytes. This call does not check for a close message marker. It may consume the close message marker and may cause an error.
receive_limited_block()	Receive a block of an exact size in bytes. This call does not check for a close message marker. It may consume the close message marker and may cause an error.

Spooling Data

Spooling takes advantage of background processes to send and receive data in cell programs. This allows a cell program to send data across the network without having to wait for the data transaction to complete. Within a cell, the computation agent starts the data movement operation. The communication agent takes over and completes the transaction. This frees the computation agent to continue with other processing. Therefore, the actual data movement takes place in the background. This lets a cell program overlap computation and communication.

A typical spooling transaction has a sending and receiving cell. The sending cell program sends data as follows:

1. sends an open message
2. starts the spool
3. sends the data
4. waits for the data to be spooled
5. sends a close message

While waiting for the spool to complete, the sending cell program can do other computations.

The receiving cell program receives data as follows:

1. receives an open message
2. starts the spool
3. receives the data
4. waits for the receive operation to complete
5. receives a close message

While waiting for the spool to complete, the receiving cell program can do other computations.

Sending Data in the Background

You can send a data block in the background using the following call:

send_block_bg()	Use a background process to send a data block to a destination cell. Otherwise, the call returns an error. This call spools the block and returns. The background process sends the block. An error occurs if the port or gate used by this call is accessed by another process during the background send process.

Use the await_bg_complete call to make sure a spool operation is completed.

Receiving Background Messages

You can receive a data block that was sent in the background using the following call:

receive_block_bg()	Receive a block of data through a spool gate. Spooling runs in the background. The call returns before the block is received. This call assumes the message path is open.

Use the await_bg_complete call to make sure a spool operation is completed.

Previewing Data

When a cell program receives data, the receive operation removes the data from the network. Before receiving data, you can use the following preview calls to view the data without consuming it.

preview_open_msg()	Preview an open message.
preview_close_msg()	Preview a close message.
preview_marker()	Preview a marker item.
preview_int()	Preview a 4-byte item of type integer.
preview_float()	Preview a 4-byte item of type float.

Each preview call is for a specific kind of data item.

The preview_open_msg(), preview_close_msg(), and preview_marker() calls return TRUE if the type of the available data item matches the type in the call, and FALSE if an incorrect data item returns.

The preview_int() and preview_float() calls return a copy of the available data as an integer or float depending on the type of the call.

If no data item is available, the preview calls wait until a data item is available.

Using Express Mode

There are two network modes: local and express. Local mode has the following characteristics:

- the computation agent controls the network interaction of data on the network
- all network data stops at the cell
- the computation agent can send and receive data on the network

In express mode, data passes through the cell without interaction with the cell's computation agent. Local mode is the default mode for point-to-point networks. For multiple sender-receiver networks, local mode is the default mode for the end cells; intermediate cells must use express mode.

When a cell receives a message addressed to it, it automatically switches to local mode. Sending or receiving a close message marker automatically switches it back to express mode.

Use the following calls to control express mode in your cell program.

skip_msg()	Set stop condition and change network mode for the current cell to express mode. This happens even if the network is in the middle of sending a message. You specify an input and output port to use for express operations. If the set up is successful, the call returns TRUE. Otherwise, the call returns FALSE.
set_stop_condition()	Set stop condition for the network associated with an input port without changing the network mode. Use this call instead of skip_msg if the network mode is already set to express mode.
get_stop_condition()	Return the stop condition set for the network associated with an input port.

Stop Conditions

Stop conditions specify when the network mode is set back to local mode for a cell. When one of the stop conditions is found by the communication agent, it switches the network mode to local mode.

PCS_STOP_CC	Stop on close connection.
PCS_STOP_CM	Stop on close message.
PCS_STOP_NM	Stop when not in a message.
PCS_STOP_OM	Stop on an open message.
PCS_STOP_SM	Stop on a software mark.
PCS_STOP_ALL	Stop on any stop condition. This sets all stop conditions.
PCS_STOP_NONE	Stop on no condition.

Waiting for Network Operations

Cell programs sometimes need to wait for network operations to complete. For example, a cell program may have to wait for a spool to complete. When using the receive_block_bg call, a cell program waits before operating on the data received. Also, sending messages on the network requires waiting until the network is available. For example, using the skip_msg call requires waiting for the network to be available.

The await calls wait for conditions to be true on the network associated with a port. When the condition is met, the call returns.

Use the following calls to synchronize data movement operations:

await_open_msg()	Wait for an open message header.
await_close_msg()	Wait for a close message trailer.
await_marker()	Wait for a marker.
await_stopped()	Wait for a stop condition to be met. For a list of the stop conditions, see page 5-18.
await_safe_for_output()	Wait until it is safe to output to the network. Use this call before a cell program writes to a network that has stopped because of stop condition instead of an open message. In this case, the cell program must wait for the network to switch from express mode to local mode.
await_network_available()	Wait until the network is available. Returns TRUE if the wait operation was successful and FALSE if the wait operation was unsuccessful.
await_bg_complete()	Wait until a background job is complete. Returns the number of bytes transferred by the spool operation.

Testing Network Status

To set up data communications, a cell program often needs to test the network status. You can either test the general network status or specific network status. Test calls test status at a port and return. They do not wait for a condition to be true.

Testing General Status

This section describes the call for making a general test of the network status, and describes the possible status values. You use the following call to test general network status.

net_status()	Test the network for status and return a status value.

The possible return values are as follows:

PCS_SKIP	Network associated with a port is in express mode.
PCS_OM	An open message marker is at the port.
PCS_SM	A software marker is at the port.
PCS_CM	A close message marker is at the port.
PCS_DATA	Data is at the port.
PCS_EMPTY	No data is available.
FALSE	Incorrect arguments.

Testing Specific Status

You use the following calls to test the network for a specific status.

test_status_open_msg()	Test the network for an open message marker.
test_status_close_msg()	Test the network for a close message marker.
test_status_marker()	Test the network for a software marker.
test_status_stopped()	Test the network to determine if the network is stopped.
test_status_data_available()	Test the network to determine if data is available.
test_status_bg_complete()	Test the network to determine if the background job is complete.

Using Dynamic Networks

In addition to creating networks at compile time, you can create dynamic networks at runtime. Dynamic networks let you create and destroy networks in cell programs, create new networks by joining existing networks into one larger network, and create multi-drop networks. You can use the send_open_network call to create point-to-point networks. When you are done using a network, destroy the network to release the network's resources.

Create a dynamic network by doing the following:

1. The source cell program sends an open network marker to a destination cell.
2. The destination cell receives the open network marker from the source cell.

Before creating a dynamic network, be sure to reserve enough logical channels for the network ports in the array program.

After creating a dynamic network, the source cell can send data to the destination cell. The destination cell can receive data.

After using a dynamic network, destroy the dynamic network by doing the following:

1. The source cell program sends a close network marker to a destination cell.
2. The destination cell receives a close network marker from the source cell.

Use the following calls to create and destroy dynamic networks:

send_open_network()	Send an open network marker with routing to a destination cell. Returns a pointer to an output port on a new network when successful, otherwise, it returns NULL.
receive_open_network()	Receive an open network marker opening a network. Returns a pointer to an input port on the network when successful, otherwise, it returns NULL.
send_close_network()	Send a close network marker over a port to close a network. Returns TRUE if the operation is successful and FALSE if unsuccessful.
receive_close_network()	Receive a close network marker closing a network. Returns TRUE if the operation is successful and FALSE if unsuccessful.

Joining Networks

During runtime, a cell program can create a new network by joining two existing networks. Use the following call to join networks together at runtime.

join_network()	Join two existing networks to create one new network. Returns TRUE if the operation is successful and FALSE if unsuccessful.

Using the Array Combiner Library

The PCS array combiner library provides the calls you use in array programs to create static networks and cell program mappings. Static networks are networks that you create at compile time, and do not change at runtime. The PCS array combiner uses the network and cell program information in the array program to build array modules.

You include the *pcs/cb_def.h* file in the source code of your array program to access the array combiner library calls.

Reserving Logical Channels

Logical channels are the hardware resources used by networks. Each cell must have enough logical channels reserved to support creating a network. An array program must explicitly reserve the logical channels for dynamic networks that are created when an array module runs. The array combiner automatically reserves the logical channels for the static networks it creates, when it creates an array module. You use the following call to reserve logical channels on a cell.

reserve_logical_channel()	Reserve logical channels on a cell. Reserves channels in the following directions: up, down, left, and right.

Creating Ports

Create a port on a cell as either an input port or an output port. When you create a port, it is not initially connected to a network or mapped to an external port. For point-to-point networks, you connect a port to a network when you create the network. You can also connect ports to intermediate cells to create multi-drop networks. You use the following calls to create ports.

create_input_port()	Create a named input port on a cell and return a pointer to an input port.
create_output_port()	Create a named output port on a cell and return a pointer to an output port.

Creating Networks

Follow these steps when creating a network:

1. Create the ports on the cell. See the section “Creating Ports.”
2. Specify a network route.
3. Create the network.
4. Connect the ports to the network for the intermediate network cells that you want to have access to the network.

The following call creates a network:

create_network()	Create a network between two ports and return a pointer to the new network. Before using this call, you need to build a network route or use the default route.

Creating Network Routes

The following calls create network routes:

make_route()	Add a cell to the beginning of a route. A route is a list of cell IDs for a network.
make_directed_route()	Add a cell, in a specific direction, to the beginning of a route.
add_to_route()	Add a cell to the end of a route.
add_to_directed_route()	Add a cell, in a specific direction, to the end of a route.
join_network()	Create a new network out of two existing networks.

There are directed and undirected versions of the routing calls. You use the directed calls to specify a direction when creating a route on a torus, because each turn in the torus can go in two directions. If the direction is not specified, the array combiner chooses the direction that results in the shortest distance to the next cell in the list.

Connecting Ports to Networks

The following calls are for connecting ports to a network:

connect_port()	Connect a port to a network. This allows the port access to the network. The network must pass through the port's cell.
get_external_port()	Return a pointer to a specified external port in an array module. Use this call to retrieve an external port from another array module.
map_external_port()	Map a port to an external port in the current array module. Use this call to provide a port to another array module.

Placing Cell Programs

You use the following calls to place cell programs on a cell.

place()	Place a cell program onto a cell specified by a cell ID. The default cell ID is cell(0,0).

You can place multiple object files on a cell. These files are linked together to create one cell program.

Combining Existing Array Modules

You use the following call to combine existing array modules with the array module you are creating.

overlay()	Overlay an existing array module onto the current array module.
create_instance()	Creates an instance of an existing array module at a specific offset, using the specified array module name. Use this call to nest array modules.

PCS Example

This example is for a two-cell application that runs on an iWarp array. There are three source code files:

- the array program source file (2cell.c)
- the cell program source file for cell(0,0) (2cell00.c)
- the cell program source file for cell(0,1) (2cell01.c)

Figure 5-8 shows the network configuration of the two-cell application.

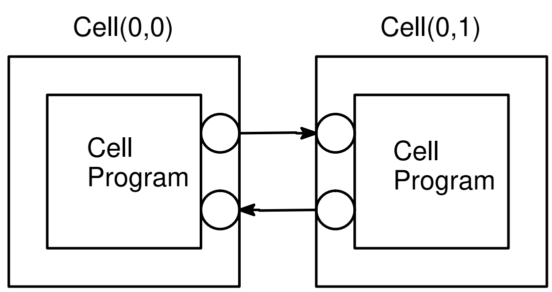

Figure 5-8: Two-Cell Application

The two-cell application consists of two cells connected by two networks. One network sends data from cell(0,0) to cell(0,1), and the other network sends data from cell(0,1) to cell(0,0).

Array Program Source File

The array program for the two-cell program does the following:

- sets the cell IDs for the source and destination ports
- creates the ports
- creates the networks
- reserves the logical channels for a dynamic network
- places the cell programs on the cells

The following source code is for the array program.

```
#include <pcs/cb_def.h>
#include <stdio.h>

user_main()
{
  cellid_type src, dst;
  network_type *nw;
  port_type *src_port, *dst_port;

  /* Configure cell IDs */
  src.type = dst.type = ARRAY;
  src.col = src.row = 0;
  dst.row = 0; dst.col = 1;
```

```
  /* Create the output port on cell(0,0) and input ports
     for cell(0,1).  Create the network between them. */
  if (!(src_port = create_output_port("Xout", &src)))
  {
    fprintf(stderr, "Output port create failed\n");
    exit(1);
  }
  if (!(dst_port = create_input_port("Xin", &dst)))
  {
    fprintf(stderr, "Input port create failed\n");
    exit(1);
  }
  if (!(nw = create_network(src_port, dst_port, NULL)))
  {
    fprintf(stderr, "Network create failed\n");
  exit(1);
  }

  /* Create the output on cell(0,1) and input ports
     for cell(0,0).  Create the network between them. */
  if (!(src_port = create_output_port("Xout", &dst)))
  {
    fprintf(stderr, "Output port create failed\n");
    exit(1);
  }
  if (!(dst_port = create_input_port("Xin", &src)))
  {
    fprintf(stderr, "Input port create failed\n");
    exit(1);
  } if (!(nw = create_network(src_port, dst_port, NULL)))
  {
    fprintf(stderr, "Network creation failed\n");
    exit(1);
  }

  /* Reserve the logical channels.  */
  reserve_logical_channel(&src, 0, 1, 0, 0, 1, PCS_POOL0);

  /* Place the cell programs on the cells. */
  place(&src, "2cell00.o");
  place(&dst, "2cell01.o");
}
```

Cell Program Source Code for Cell(0,0)

The cell program for cell(0,0) of the two-cell program does the following:

- initializes the networks and ports
- binds the systolic gates to ports to set up systolic data movement on the network
- opens and tests the network from cell(0,0) to cell(0,1)
- creates a dynamic network to cell(0,1) and sends a message block
- sends a message to cell(0,1)
- closes the networks and exits

The following source code is for the cell program that runs on cell(0,0).

```
#include <stdio.h>
#include <pcs/pcs_def.h>
#include <pcs/pcs_asm.h>
#define DATA_NUM 16 port_type *Xout, *Xin;


port_type *new_port;
int address[DATA_NUM];
port_list_type ports[] = {{ "Xout", &Xout },{ "Xin", &Xin }};


main()
{
  cellid_type dst;
  int i;
  cellid_type *self;
  int count;

  /* Initialize the port connections. */
  pcs_init(ports, NUM_PORTS(ports), NULL, 0);
  dst.row = 0;  dst.col = 1; dst.type = ARRAY;
  for (i = 0; i < DATA_NUM; i++)
    address[i] = i;

  /* Bind systolic gates to ports */
  bind_systolic_gate(Xout, GATE0);
  bind_systolic_gate(Xin, GATE0);

  /* Send messages and test the network to cell(0,1).*/
  send_open_msg(Xout, GATE0, NULL);
  receive_open_msg(Xin, GATE0);
  send_int(Xout, GATE0, 1);
  receive_int(Xin, GATE0);
  send_close_msg(Xout, GATE0);
  receive_close_msg(Xin, GATE0);


  /* Create a new dynamic network. */
  if (!(new_port = send_open_network(GATE1, &dst, (NULL))))
    printf("cannot open connection\n");
```

```
    /* Send a message to cell(0,1). */
    if (!send_direct_msg(new_port, GATE1,
         (char *)address, sizeof(int)*2, NULL))
      printf("cannot send message\n");

    /* Close the dynamic network. */
    if (!send_close_network(new_port, GATE1))
      printf("cannot close connection\n");

    /* Send a message through first network. */
    for (count = 0; count < 1000; count++)
    {
      if (!send_direct_msg(Xout, GATE0, address,
        sizeof(int)*DATA_NUM, NULL))
      printf("cannot send direct message\n");
    }
    pcs_de_init();
     exit(0);
}
```

Cell Program Source Code for Cell(0,1)

The cell program for cell(0,1) of the two-cell program does the following:

- initializes the networks and ports
- binds the systolic gates to ports for systolic data movement on the network
- opens and tests the network to cell(0,0)
- creates a dynamic network to cell(0,0) and receives a message block
- receives a message block through GATE0 from cell(0,0)
- closes the networks and exits

The following source code is for the cell program that runs on cell(0,1).

```
#include <stdio.h>
#include <pcs/pcs_def.h>
#include <pcs/pcs_includes.h>
#include <pcs/pcs_asm.h>
#define DATA_NUM 16

int address[DATA_NUM];
port_type *new_port;
port_type *Xin, *Xout;
port_list_type ports[] = {{ "Xin", &Xin }, { "Xout", &Xout}};

main(argc, argv)
int argc;
char *argv[];
{
  int i;
```

```
  int num;
  cellid_type *self;
  cellid_type dst;
  int count;
  char *char_addr;


  /* Initialize the ports and gates. */
  pcs_init(ports, NUM_PORTS(ports), NULL, 0);
  dst.row = dst.col = 0; dst.type = ARRAY;
  bind_systolic_gate(Xout, GATE0);
  bind_systolic_gate(Xin, GATE0);


  /* Open a message and test the network to cell(0,0). */
  if (!send_open_msg(Xout, GATE0, NULL))
     printf("cannot send open msg");
  if (!receive_open_msg(Xin, GATE0))
    printf("cannot receive OM");
  send_int(Xout, GATE0, 1);
  receive_int(Xin, GATE0);
  if (!send_close_msg(Xout, GATE0))
     printf("cannot send close message");
  if (!receive_close_msg(Xin, GATE0))
    printf("cannot receive open message");

  /* Creates a dynamic network with cell(0,0). */
  /* Receive a message, then close the network. */
  if (!(new_port = receive_open_network(GATE1)))
     printf("cannot open connection");
  if (!(num = receive_limited_msg(
      new_port, GATE1, address, sizeof(int)*2)))
    printf("cannot receive message");

  /* Close the dynamic network. */
  if (!receive_close_network(new_port, GATE1))
    printf("Cannot close network");

  for (i = 0; i < DATA_NUM; i++) address[i] = 0;

  /* Receive a message from the first network. */
  for (count = 0; count < 1000; count++)
  if (!(num = receive_limited_msg(
    Xin, GATE0, (char *)address,
    sizeof(int)*DATA_NUM)))
    printf("cannot receive message");

  pcs_de_init();

  exit(0);
}
```